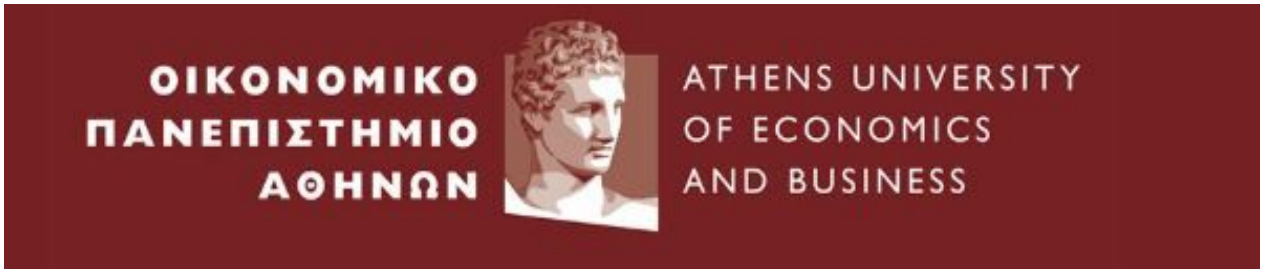




AUEB M.S.c in Data Science (part-time)

# Text Analytics: Homework 3: Report

## Team members:

**Kaplanis Alexandros (P3351802)**

**Politis Spiros (P3351814)**

**Proimakis Manos (P3351815)**

# 1. SETUP & EXECUTION

Please refer to the README.md for instructions.

# 2. INTRODUCTION

We opted for reusing the dataset that was employed in our last assignment, namely the Twitter Sentiment dataset which has already been downloaded from Kaggle for the Sentiment140 project. We have already performed significant work on data preparation and word embeddings, therefore we consider our previous effort as a good baseline from which to start. As a reminder, the datasets targets 2 classes; 0 for negative and 4 for positive, which are equiprobable with 800,000 records per class.

## 2.1 SCOPE & GOAL

In our current iteration our goal is to train an MLP implemented in Keras and tune the model to have good prediction score as well as good generalization. In our previous attempt we have chosen a tuned *Logistic Regression* model using *TF/IDF* vectors, which achieve 0.77 f1 score on the unseen data. This is a pretty good score and given the computation power that a MLP requires to be trained and tune(i.e. the use of more data), it would be good to verify if it would provide any better outcome than a simple LR.

## 2.2 EMBEDDINGS

Again we decided to experiment with both *TF/IDF* and W2V embeddings to train our MLP. In the previous assignment we had a clear indication that *TF/IDF* worked better for classification as we had better accuracy on the validation set.

We experimented with some base models using *W2V* embeddings and the results where lower than using *TF/IDF* vectors for the same model. Therefore, we decided to drop *W2V* vectors altogether, in favor of *TF/IDF* ones.

Note that we were able to increase the number of training instances from ~50.000 to ~200.000, compared to Assignment 2, mainly because the neural network architecture, in combination with GPU computation, allows for the inclusion of more training data.

# 3. MLP Architecture

As a first step we decided to have a look on which parameters play a key role in the accuracy of our model. We didn't want to use any automation on this step, as it is very time consuming, and it would be very important to tune only the parameters that would help increase the efficiency.

On the other hand, manual tuning does not allow us to see every possible combination of the parameters. As a result we experimented by tuning one parameter at a time, noted which were the best performing values and then moved on to the next parameter. This way we got a good grasp on which parameter values work for our model which will be further tested in the hyperparameter tuning phase with Talos.

## 3.1 SHAPE

After taking into consideration our knowledge on MLP and collecting much information, generally an MLP with 1 layer is able to approximate any linear transformation from one finite space to another and an MLP with two hidden layers is sufficient for creating classification regions of any desired shape.

In practise we can only have an intuition of the shape of the model but we can only evaluate it with trial and error. On the other hand what we can do is efficiently set the number of neurons of each layer as well as other parameters that will help us fit (but not overfit) the best model with a predefined shape. We have used *TF/IDF* vectors of 15000 features on our vectorizer, so our input dimension will have this size, and the output layer will have 1 (sigmoid or tanh) activation to transform the out to a probability.

On our initial exploration step we have used early stopping in Keras, which prevents a model from overfitting by watching the loss on the holdout set, essentially training will be stopped when the model starts to lose its generalization ability by miss-predicting the validation set. A good approach to this, is to use a broad starting network and evaluate how and if additional epochs help the model predict the validation set.

## 3.2 OBSERVATIONS

### 3.2.1 ACTIVATION FUNCTIONS

The first parameter we chose to examine was the activation function. A first try was with no function at all. The results were disappointing but it was expected as we don't perceive our problem to be linearly solvable. Proceeding with the examination we tried using several activation functions in both the hidden and the output layers. The conclusion was that since our output yields only two values we should either use the sigmoid or tanh functions. As for the hidden layers, either of the *softmax*, *tanh* or *relu* would produce satisfying results which would vary depending on the rest of the parameters.

### 3.2.2 NEURON SIZE & REGULARIZATION

Neuron size, along with epochs and regularization techniques, are crucial in terms of the bias-variance trade-off. Using a small neuron size would lead to underfitting while an unnecessarily big size to overfitting. As for regularization we used the already mentioned early stopping and dropout methods. Dropout would produce varying degrees of results depending on the rate used. A large rate will introduce more bias to the model and possibly lead to underfitting while a low rate will simply reduce variance a little. Our baseline model seemed to work better for values in the [0.2, 0.5] range which could mean that we have enough bias in our model and need some variance reduction.

### 3.2.3 OPTIMIZERS

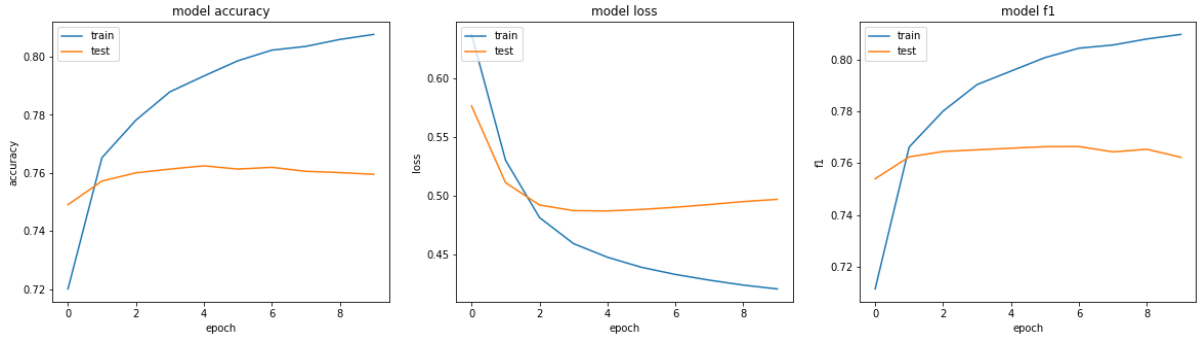
Last but not least come the optimizers. The four optimizers we tried were *SGD*, *Adagrad*, *Adadelta* and *Adam*. Both *Adagrad* and *SGD* were soon dismissed. *Adagrad* was out because our data are not sparse and the decaying learning rate of the algorithm would lead to a slightly slower convergence than with the other two adaptive learning-method algorithms.

*SGD* would at first output a constant value for loss with no differentiations through the epochs. Based on theory *SGD* is sensitive to saddle points, and as a result we tried using momentum which produced great fluctuation to our loss results, meaning possibly that it accidentally bypassed the minima. As a final solution we tried using the *Nesterov* method on top of the momentum, but the results were only slightly better than before, leading to the dismiss of this optimizer.

In the end, the optimizer of our choice was *Adam*. While *Adadelta* would surpass *Adagrad's* issue and produce satisfying results, *Adam* would outperform all other algorithms.

### 3.2.4 CONCLUSION

The best base model was selected based mainly on the performance of the model (*F1* score) on the dev set. In order to decide, we observed all manual executions of the models and selected the one with the best *F1 score*. The best model's graph is shown below:




In general, we observed that throughout our experiments the *F1* score was beginning to decrease after 4-5 epochs. A complete picture of the iterations is present in our Jupyter Notebook.

The work described above was done in order to establish a base set of hyperparameters in accordance with theory. Subsequently, we further optimized this set, performing hyperparameter optimization with Talos.

# 4. HYPERPARAMETER OPTIMIZATION

During the hyperparameter tuning phase we considered several options. Some of the most widely used approaches we had in mind are the following:

- *Baby-sitting (Trial & Error)*: a manual process in which model run and evaluation is done by hand-coding the parameters and running the experiment (rejected). We followed this approach for establishing the base model, for which subsequently we performed parameter tuning.
- *Grid Search*: an approach in which all combinations of a fixed set of parameters are evaluated on the model. Quite an expensive process when combined with Deep Learning, which we avoided.
- *Random Search*: employes sampling from a configuration search space in which hyperparameters are sampled from a random pool. We employed this method in our hyperparameter optimization.

- *Bayesian Optimization*: the most sophisticated method, based on a surrogate model whose aim is to learn the optimal hyperparameter mix, using Bayesian processes. This method was eventually deemed as out of scope for this assignment.

## 4.1 IMPLEMENTATION DETAILS

In order to search our hyperparameter space, we opted for using Talos, an open-source hyperparameter optimization package known to be working well with Keras.

## 4.2 METHODOLOGY

After having established a base model, we performed hyperparameter tuning with *Talos* on the selected model. The selection criterion was based on the Talos automatic best-model selection mechanism, which employees the *F1* score.

We chose to search for the following mix:

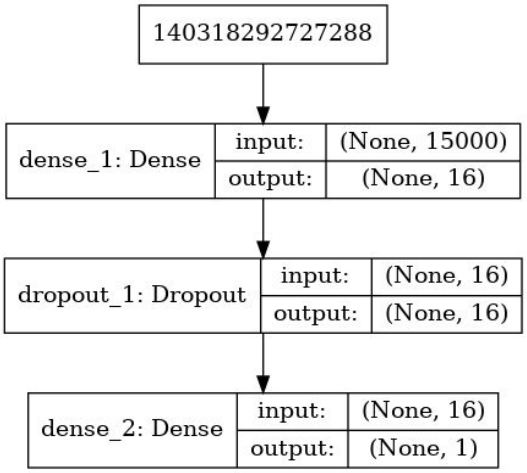
- 1 or 2 hidden layers
- Different numbers of neurons for the first layer (16, 32, 64) and the second layer (8, 16, 32)
- A batch size of 100 or 200
- Training for 20 or 50 epochs (in most cases, early stopping prevented iteration through all epochs)
- Dropout rate of 0.2, 0.3 or 0.5
- Different values for the Learning Rate (1e-3 or0.01)

The hyperparameters search space is in Python dictionary format as required by Talos, is shown below:

```
hyperparameters = {
   'learning_rate': [1e-3, 0.01],
   'first_layer_num_neurons':[16, 32, 64],
   'second_layer_num_neurons':['inactive', 8, 16, 32],
   'first_layer_activation': ['relu'],
   'second_layer_activation': ['relu'],
   'batch_size': [100, 200],
   'epochs': [20, 50],
   'dropout': [0.2, 0.3, 0.5],
   'optimizer': [Adam],
   'last_layer_activation_function': ['sigmoid'],
   'weight_regulizer': [None]
}
```

# 5. RESULTS

The final architecture of our MLP is shown below:




Before using the *held-out* dataset we normalized it, meaning that we applied the same preprocess as with the training data. The resulting *F1* score was 0.82 and the confusion matrix of the unseen dataset was the below:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.82 | 0.81 | 0.81 | 177 |
| 1 | 0.82 | 0.83 | 0.82 | 182 |
| accuracy | | | 0.82 | 359 |
| macro avg | 0.82 | 0.82 | 0.82 | 359 |
| weighted avg | 0.82 | 0.82 | 0.82 | 359 |

| Predicted | 0 | 1 |
|---|---|---|
| True | | |
| 0 | 143 | 34 |
| 1 | 31 | 151 |

The final classification gives a good ratio between the *TN-FN* (5/2) and *TP-FP* (5/1) so we have an overall efficient classifier. Some random examples of misclassification are depicted below:

| | actual | predicted | text |
|---|---|---|---|
| 26 | 0 | 1 | rt new time warner slogan time warner where we make you long for the days before cable |
| 27 | 0 | 1 | i still love my kindle2 but reading the new york times on it does not feel natural i miss the bloomingdale ads |
| 28 | 0 | 1 | although today s keynote rocked for every great announcement at amp t shit on us just a little bit more |
| 29 | 0 | 1 | fuzzball is more fun than at amp t p ball com twitter |
| 36 | 1 | 0 | lebron best athlete of our generation if not all time basketball related i don t want to get into inter sport debates about $\frac{}{2}$ |
| 37 | 1 | 0 | lebron is a beast nobody in the nba comes even close |
| 38 | 1 | 0 | good news just had a call from the visa office saying everything is fine what a relief i am sick of scams out there stealing |
| 39 | 1 | 0 | rt i love the nerdy stanford human biology videos makes me miss school |

From the above results we can see that the tweets that our classifier failed to predict correctly are very obscure and are really hard even for a human to understand the correct sentiment. (We have the full list of misclassified tweets in the notebook).

In conclusion, the model we decided to proceed with was a fairly simple model. It turned out that only one hidden layer and 16 nodes would produce good enough results which were in fact better than the results of our previous logistic regression model.